Section 23
Traffic Model Design Overview

This section includes explanations of the aircraft profiling techniques and inter-process communications that are used throughout the three Traffic Model Functions: the *Parser*, the *Flight Database Processor*, and the *Traffic Demands Database Processor*.

23.1 Aircraft Dynamics Modeling

Purpose

The speed and altitude of an F16 are obviously quite different from those of a B737 during the takeoff and climb phases. Similarly, different aircraft types may differ in speed in the descent phases. This section describes a mathematical model that accurately predicts the speed, altitude, flight phase, and time along the flight path and accounts for the type of aircraft. There are three major routines in this model; all use information from an associated **aircraft dynamics database**.

Two of these routines are accomplished in the *Parser*. The first is appropriately named *assign_a_profile*. For each flight, the *Parser* calls this routine to initialize many of the flight variables. This routine has two purposes:

- To assign an ascent and a descent profile.
- To check for inconsistencies and errors in the flight variables in the NAS messages given the type of aircraft.

The second routine is repeatedly called by the *Parser* throughout the entire flight. Its function is accomplished by a routine aptly named *get altitude value*, whose purpose is to predict the altitude, speed, and flight phase for a variety of aircraft along any reasonable flight path. For details on this routine, see Section 24.

The third routine is called *get time value*. It is repeatedly called by the *Flight Database Processor* later in the overall processing. The purpose of this routine is to predict the time for each flight event for a flight. For details on *get time value*, see Section 25.

Figure 23-1 depicts a top level data flow diagram showing the essential aircraft dynamics modeling elements: functions, data storages, and data flows. These elements are described in more depth in later sections.

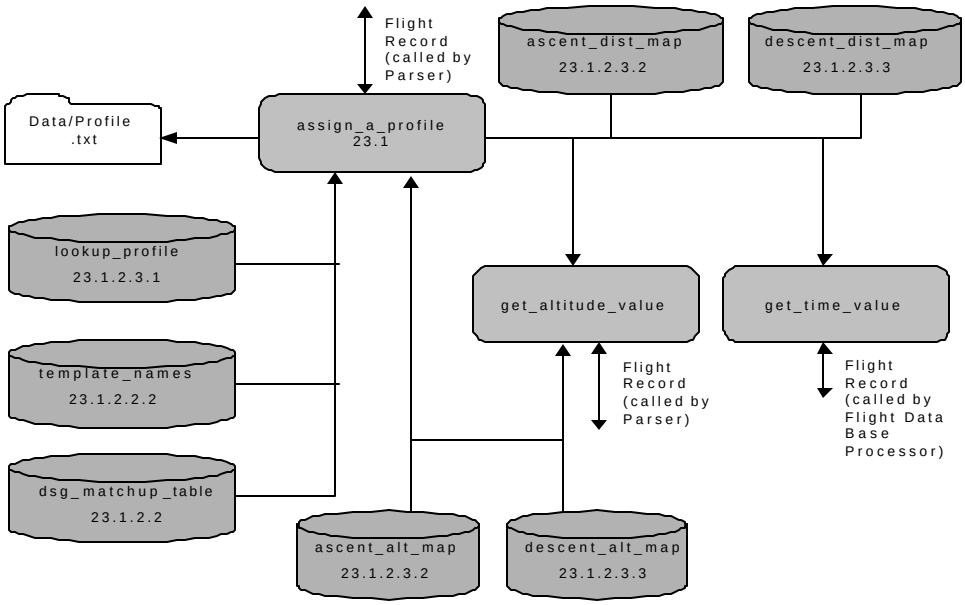

Figure 23-1. Aircraft Dynamics Data Flow Diagram

23.1.1 The Flight Record: a Common Vehicle for I/O

All three routines (*assign_a_profile, get_altitude_value, and get_time_value*) previously described pass information from the calling procedures using the **flight** record. While the flight record structure does not vary from routine to routine, the usage does change. For instance, field name values produced as output from the *assign_a_profile* procedure may be used as input to the *get_altitude_value* procedure. Similarly, **flight** record fields functioning as output in the *get_altitude_value* procedure will be used as input in the *get_time_valu*e procedure.

In addition to a variety of fields, there are two small records embedded in the **flight** record.

The two are identical in structure and contain the following variables: distance from takeoff, altitude, speed, latitude, longitude, time, and phase. As the flight is developing, the first sub-record, referred to as the **previous** node, contains the values of the previous position of the aircraft (i.e., the previous procedure call). The values of this sub-record are always available as input. The second sub-record, named the **now** node, contains the values of the current procedure call and the input/output usage of the fields differs depending on the procedure. Upon returning from any of the profile procedures, the calling procedure must move the **now** node values to the **previous** node and prepare the **now** node input fields for the next call. The elements (sub-records, field names, etc.) of the **flight** record are described in Table 23-1.

NOTE: The speed is in units of 100*(nautical miles/minute). All the other elements are in their usual units.

Table 23-1. Aircraft Dynamics Flight Record

Flight Record							
Library Name: Profile_openlib		**Element Name:** Profile.h					
Data Item	**Definition**	**Unit**	**Legal Range**	**Var. Type**	**I/O by Function +**		
					F_1	**F_2**	**F_3**
flt_id	Flight identifier (e.g. AAL 123)			array [1..10] of char	I	-	-
filed_fz_onground	indicates the disposition of the filed Field10 Field 10 filed on ground = T; filed in air = F		T or F	boolean	I	-	-
civ	indicates if the aircraft is civilian or military civilian aircraft = T; military aircraft = F		T or F	boolean	O	I	-
runaway	indicates that flight errors or inconsistencies are severe & fatal = T; no major problem = F		T or F	boolean	O	-	-
dsg-actual	actual designator taken from the FZ			array [1..4] of char	I	-	-
aircraft_index	index indicating a record in the Aircraft_Description Map which describes the given aircraft flight		-1 to max_plane_type	short	O	-	-
dsg_index	index indicating the particular template aircraft assigned to this flight		1 to tot_templates	short	O	I	I
ascent_index	index indicating the particular ascent profile for this flight		1 to max_ascent_profile	short	O	I	I
descent_index	index indicating the particular descent profile for this flight		1 to max_descent_profile	short	O	I	I
dist_total	total distance for this flight	n.miles		INT32	I	I	I
origin_lat	latitude of the originating airport	radians		float	I	I	I
origin_lon	longitude of the originating airport	radians		float	I	I	I

Flight Record (cont'd)							
Library Name: Profile_openlib		**Element Name:** Profile.h					
Data Item	**Definition**	**Unit**	**Legal Range**	**Var. Type**	**I/O by Function +**		
					F_1	**F_2**	**F_3**
dest_lat	latitude of the destination airport	radians		float	I	I	I
dest_lon	longitude of the destination airport	radian		float	I	I	I
dist_cruz	distance from the takeoff roll to the point at which the aircraft achieves crusing altitude	n.miles		INT32	O	I	-
spd_cruz	crusing speed for this flight	(n.miles/ min)x100		INT32	I/O	I	I
alt_cruz	crusing altitude for this flight	feet/100	0 to 600	INT32	I/O	I	I
dist_descent	distrance from the begin descent point to the point where the aircraft touches down	n.miles	0 to 600	INT32	O	I	I
previous.dist	previous distance along the flight path	n.miles		INT32	O	I	I
previous.lat	latitude at the previous location for this flight	radian		float	O	I	I
previous.lon	longitude at the previous location for this flight	radian		float	O	I	I
previous.phase	flight phase at the previous location for this flight			flight phase see 7.1.2.2			

Table 23-1. Aircraft Dynamics Flight Record (continued)

Notes: + F_1 *indicates* Assign_A_Porfile, F_2 *indicates* Get_Altitude_Values, F_3 *indicates* Get_Time_Vaule

Table 23-1. Aircraft Dynamics Flight Record (continued)

Flight Record (cont'd)							
Library Name: Profile_openlib		**Element Name:** Profile.h					
Data Item	**Definition**	**Unit**	**Legal Range**	**Var. Type**	**I/O by Function +**		
					F_1	**F_2**	**F_3**
previous.alt	altitude at the previous location for this flight	feet/100	0 to 600	integer32	O	I	I
previous.speed	speed at the previous location for this flight	(n.mile/ min)x100		integer32	O	I	I
previous.time	accumulated time from takeoff roll to the previous location of the flight	minutes		real	-	-	I
now.dist	current distance along the flight path	n.miles		integer32	I	I	I
now.lat	latitude at the current location for this flight	radian		real	I	I	I
now.lon	longitude at the current location for this flight			real	I	I	I
now.phase	flight phase at the current location for this flight			flight phase see 7.1.2.2	O	O	I
now.alt	altitude at the current location for this flight	feet/100	0 to 600	integer32	O	O	I
now.speed	speed at the current location from this flight	(n.miles/ min)x100		integer32	O	O	I/O
now.time	flying time from the previous location to the current location	minutes		real	-	-	O
no_descent	indicated whether or not the descent is modeled no_descent modeled = T; not land now = F		T or F	boolean	1	1	1
get_down	indicates whether the flight must land on this call or not. Must land now = T; not land now = F		T or F	boolean	-	1	1

Notes: + F_1 *indicates* Assign_A_Porfile, F_2 *indicates* Get_Altitude_Values, F_3 *indicates* Get_Time_Vaule

The flight phase is a user defined type and is used in the **previous.phase** and **now.phase**. It is assumed that, for a flight, vertical movement proceeds sequentially from ascent to level flight to descent. Table 23-2 shows seven ordered values that describe the phases of flight.

Table 23-2. User Defined Flight Phases

Value	Flight Phase	Description
0	takeoff_phase	Takeoff roll up to either FL100 or cruise altitude, whichever is lower.
1	climb_phase	Ascending from FL100 to cruise altitude.
2	level_out_phase	Flying level within the originating TCA (i.e., 30 nautical miles from the original airport and below FL100).
3	enroute_phase	Flying level and not in the originating TCA.
4	arrive_phase	Descending from cruise altitude down to FL100.
5	approach_phase	Descending from FL100 down to touchdown.
6	landed_phase	Flight has landed.

The ordered nature of this type constrains the possible current values based on the previous value; the **now.phase** can be either equal to or higher than the **previous.phase**. For example, if the previous phase has a value of **enroute_phase**, the current phase may be en route, arrive, approach, or landed. Note, however, that after takeoff, a flight may either climb or level out depending on the particular flight plan; both values can never be attained for the same flight.

23.1.2 Aircraft Dynamics Database

Seven map files comprise the **aircraft dynamics database**. Each map file is accessed by a pointer. During the startup of the system, the *Parser* opens the map files and calls the *set_profile_map_ptrs* procedure (in module /**atms**/**libraries**/**profile_lib**/**profile_routines.pas**) to set the pointers to the map files. The *Parser* closes the map files as required. The map files names are: **dsg_matchup_table**, **template_names**, **lookup_profile**, **ascent_alt_map**, **ascent_dist_map**, **descent_alt_map**, and **descent_dist_map**. Each is shown in Figure 23-1; they are described in more detail in Sections 23.1.2.2 and 23.1.2.3. In addition, information about the various aircraft types is summarized in the map file **aircraft_categories.map**, which is used by the Schedule Data Base (SDB) and the *Parser.*

23.1.2.1 Error Adjustment in the Aircraft Dynamics Database

The mapped files of the database use a convention regarding the *correctness* of variable values. All of the numerical variables in the database may be physical in nature (e.g., altitude, speed, distance, or time) or may be used as an index in another array. According to our convention, a non-negative value indicates that the value is correct or consistent in the context of the database. A negative value is indicative of an error. When a negative value is detected,

the data is processed to account for the problem.

23.1.2.2 The dsg_matchup_table Map File

The **dsg_matchup_table** (Aircraft Descriptor) map file is used to characterize the aircraft employed in a given flight. It is structured as an array in which each element is a record containing the same six fields. There are currently 715 such records, each describing an aircraft model. The fields are described in Table 23-3.

Aircraft Descriptor Map File

Library Name: **profile_openlib**

Purpose:

This data structure is used to get the aircraft performance data for ascent and descent profiles, and to categorize the aircraft type, as to weight class, category, etc.

Element Name: **profile.h**

Data Item Definition Unit/Format Range Var. Type/Bits
dsg Aircraft designator from FAA Publication 7340 “Contractors” [A..Z], [0..9] string4
id Index of template aircraft most similar to this aircraft [1..44] short
group One of 7 aircraft categories, similar to aircraft category. grp user defined see below
wt_cls Weight class + S or L or H char
civ True if civilian aircraft False if military aircraft Boolean
faacat For definitions, see Table 19-13 [1..9] unsigned short

Table 23-3. dsg_matchup_table (Aircraft Descriptor) Data Structure

† Small < = 12,5000 lbs.
12,500 < Large < 300,000 lbs.
Heavy = > 300,000 lbs.

The **group** field has seven categories which grossly classify the aircraft and are shown in Table 23-4. The factors which determine the categories include weight class, type of propulsion, and typical mission (e.g., general aviation (GA), commercial, attack/fighter, etc.). The **grp** categories are similar to, but not the same as, the **faacat** field (see Table 19-13 for **acft_category** definitions).

Table 23-4. Grp Values

Value	Enumerated Name	Definition
1	PISTONPROP	Piston—propeller drive aircraft
2	TURBOPROP	Turbine—propeller drive aircraft
3	LARGE_COM_JET	Small or Large GA/commercial jet
4	HEAVY_COM_JET	Heavy GA/commercial jet
5	FIGHTER	Jet trainer, fighter, or attack aircraft
6	BIG_MIL_JET	Military jetcargo/tanker/bomberaircraft
7	HELICOPER	Helicopter (of any type)

23.1.2.2.1 The Designator-Template Matching Rule

In order to simulate the ascent of any aircraft accurately, the aircraft dynamics model first requires detailed profile data for a population of the most popular aircraft. For the purposes of this section, the popular aircraft are called *template* aircraft. There are currently 44 template aircraft in the population. This consists of 43 template aircraft from previous work on the Integrated Noise Model (see *ETMS Functional Description*, Section 7) and an additional model for helicopters.

The aircraft dynamics model then requires a rule which determines the template aircraft most similar to the aircraft being modeled. The modeled aircraft uses the profile data from the matched template aircraft to simulate the flight. With one exception, the matching rule first constrains the search for the template most similar to those that have the same **grp** category as the aircraft in the flight. If the given aircraft is identical to one of the template aircraft there is an obvious match. If not, the given aircraft is compared with each template in the category with regard to the maximum comfortable climb rate and dive rate as published in FAA Publication 7340. Among those in the appropriate **grp** category, the template aircraft that produces the smallest value computed according to the formula illustrated in Figure 23-2 is then matched to the modeled aircraft.

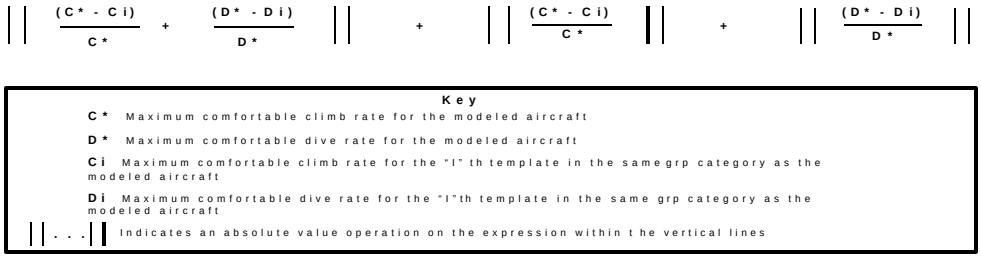

Figure 23-2 Template Aircraft Matching Formula

This rule is applied in a program named match_planes, described in Section 32, which creates a new **dsg_matchup_table** map file (as well as **template names** and **lookup_profile** map files) when a new version is required (because of changes in **template_ac** or **candidate_ac**). In routine application, when the ascent of a given flight is to be simulated, one searches in the dsg_matchup_table map file for the record containing the appropriate dsg value. The value of the ID field in that same record is used as an index to the template aircraft.

23.1.2.2.2 The template names Map File

The template names map file is used to find the name of the template aircraft when an error or inconsistency is detected by the assign_a_profile procedure. When an error or inconsistency is found, an error record is written to the data/profile.txt file. The map file is quite small and is structured as a doubly indexed array. The first index points to a particular template aircraft; there are currently 44 such templates. The second index points to any one of four characters which comprise the designator of the template. The structure of template names map is shown in Figure 23-3.

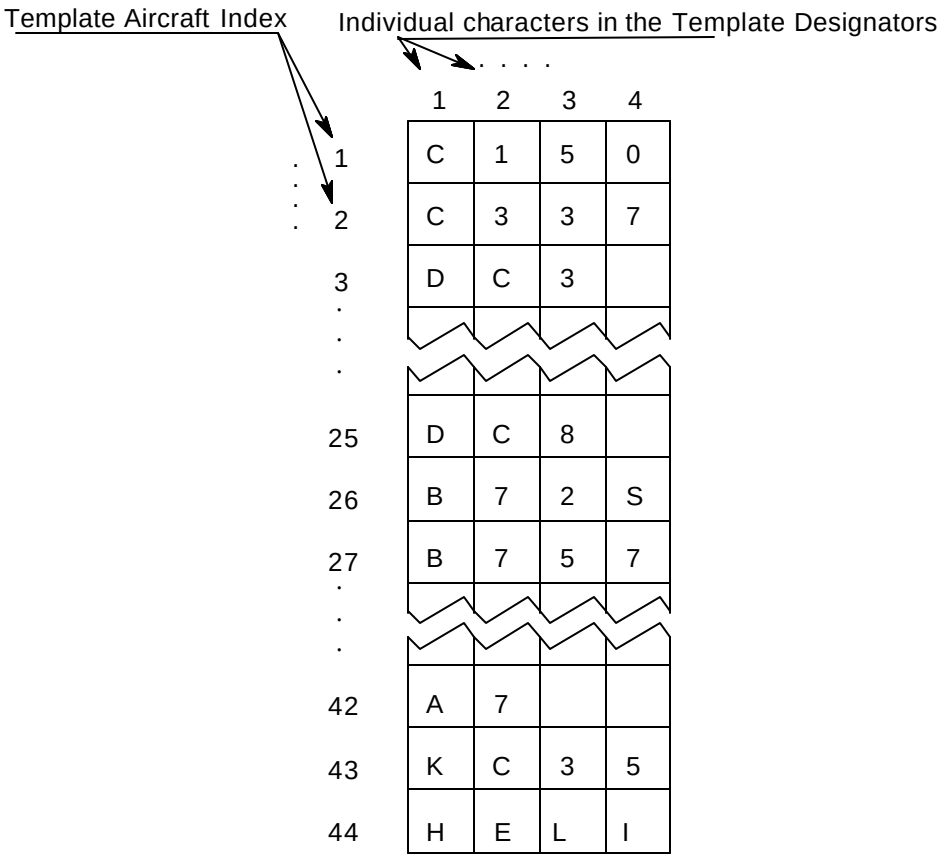

Figure 23-3. Structure of the template_names Map File

23.1.2.3 Profile Map Files

The profile map files contain the detailed data essential in modeling the ascent and descent of the flight. There are two ascent and two descent map files. The total size of the two ascent map files is approximately 256 kilobytes; the total size of the two descent map files is substantially less (approximately 9 kilobytes). The map files are structured as arrays and the elements are formatted in **INT32** allowing for both high speed and accuracy. These map files are initially created by the stand-alone program named map_profiles, described in Section 32.

23.1.2.3.1 Determining an Ascent Profile Using the lookup_profile Map File

The two ascent map files are the **ascent_alt_map** map file, and the **ascent_dist_map** map file. Each ascent map is grossly partitioned into 128 ascent profiles, and each of these profiles is associated with one or more template aircraft and one or more flight distance categories.

There are seven such distance categories: a Category 1 flight length ranges up to 500 nautical miles, while a Category 7 flight distance extends beyond 4500 nautical miles. Each intermediate category excludes all others.

A given ascent profile is addressed by an ascent index. These ascent indices are distributed in the **lookup_profile** map file. The structure of the **lookup_profile** map file is presented in Figure 23-4. The template aircraft index determines the row position of the map file while the flight length category specifies the column position of the map file. A negative value in the map file indicates that the aircraft-flight length combination is mildly inconsistent. That is, the particular template aircraft's range is too great. The inconsistency is noted and the absolute value of that element is used as an **ascent_index.** In addition, each template aircraft has a generous maximum range limit. If the total distance for a flight exceeds the limit for the flight aircraft's template, the flight is discarded on the basis that the distance is unreasonably long.

23.1.2.3.1 Ascent Map Files: Purpose and Structure

The **ascent_alt_map** map file is used to look up the distance along the ascent trajectory from the takeoff roll to a given altitude when presented with the given altitude, type of aircraft, and the total flight distance. This map file is structured as a doubly indexed array where the elements are in **integer32** format. The first index is the **ascent_index**, described in the previous sections, and encompasses both the type of aircraft and the total flight distance. Each of the 128 ascent index values is associated with an ascent profile. The second index (i.e., **alt_index**) is computed by dividing the altitude (in feet) by 1000 and then rounding off. Hence, the **alt_index** ranges from 0 to 60 (i.e., sea level to FL600).

The **ascent_dist_map** map file is used to look up the altitude, the speed, or the time from the start of the takeoff roll given the current distance along the flight path, the type of aircraft, and total flight distance. This map file is structured as a triply indexed array where the elements are in INT32 format. The aircraft type and total distance determines the value of the **ascent_index**. The second index is computed by dividing the current flight distance (up to 250 nautical miles) by 2. The third index determines which type of flight variable is used:

1 = altitude
2 = speed
3 = time.

NOTE: The speed is in units of 100*(nautical miles/minute). The other elements are in their usual units, namely altitude in hundred feet, and time in minutes.

23.1.2.3.2 Descent Map Files: Purpose and Structure

The descent map files are quite similar to the ascent map files with the obvious difference regarding the direction.

The **descent_alt_map** map file is used to look up the distance along the descent trajectory

from the touchdown back up to a given altitude when presented with the given altitude. This map file is structured as a singly indexed array where the elements are in **INT32** format. The index (i.e., **alt_index**) is computed by dividing the altitude (in feet) by 1000 and then rounding off. Hence, the **alt_index** ranges from 0 to 60 (i.e., sea level to FL600). This uses a common descent trajectory with a leveling out point at FL120.

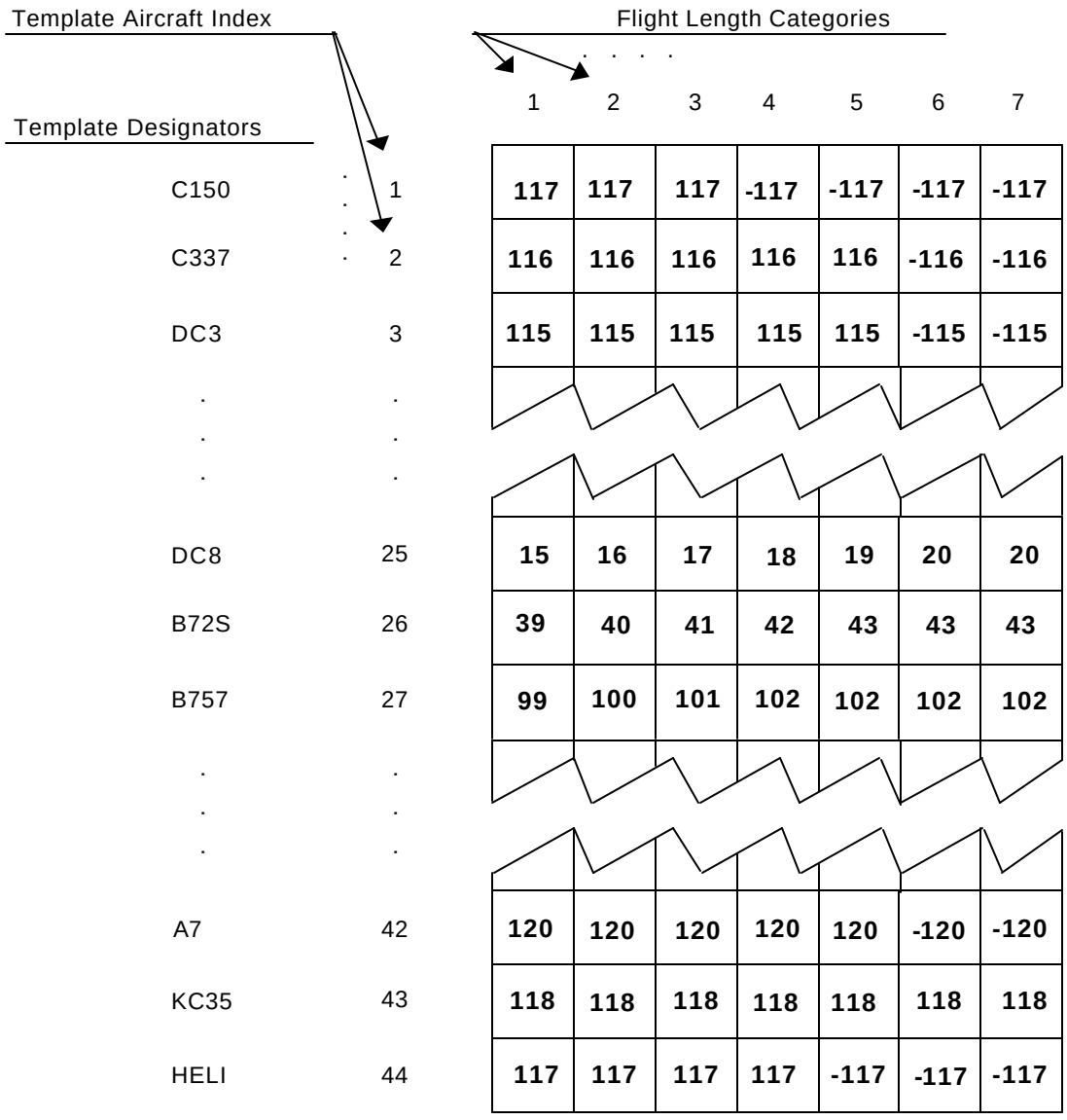

Figure 23-4. Structure of the lookup_profile Map File

The **descent_dist_map** map file is used to look up the altitude, the speed, or the time to touchdown given the current distance along the flight path, the type of aircraft, and the cruising speed and altitude. This map file is structured as a triply indexed array where the elements are in **INT32** format. The aircraft descent group and the cruise Mach number determine the value of the **descent_index**. According to the descent profile rule, an aircraft initially descends at a given constant Mach number (determined by the cruise altitude and speed) and then, at a transition altitude, follows a given indicated airspeed (IAS) to FL120. The speed is reduced to 225 knots upon entering the terminal control area, and the flight's final speed at approach is used according to the aircraft's descent group.

There are nine **descent_index** values. The first four include heavy commercial jets, fighters, and big military jets. These indices increase with decreasing Mach numbers. The next four index values include large commercial jets, and they follow the same index order and Mach number scheme. The last descent index includes all pistonprops and turboprops regardless of cruising speed. The second index is computed by dividing the current flight distance (up to 200 nautical miles) by 2. The third index determines which type of flight variable is used:

1 = altitude
2 = speed
3 = time.

23.2 Inter-process Communications

The *Generic Buffering Package (GBP*) has been developed to address the inter-process communications needs of the Traffic Model Functions. For an overview of the reasons for developing these generic routines, see Section 5.

The main requirements that have guided the development of the *GBP* are as follows:

- To optimize usage of computing resources, by isolating data processing from I/O.
- To standardize inter-process communications for all Traffic Model Function components.
- To make inter-process communications details transparent to the user, whether they are local or remote - i.e., within the same node or between nodes in a Local Area Network (LAN).
- To provide a flexible inter-process communications scheme, supporting diverse data flow configurations.
- To optimize data throughput rates across the Traffic Model Functions.

The *GBP* uses queues for inter-process communications. Due to a flexible memory

management strategy, queues can shrink or expand on demand, up to user-defined size limits. This buffering feature gives each process some degree of *asynchronicity* in an otherwise tightly coupled environment.

Design Issue: Queues as Shared Memory

Queues can be shared in a flexible manner to coordinate data flows between multiple processes. Sharing of queues ensures that data is processed in overall time sequence, while fully utilizing the data processing resources available through multiprocessing. Figure 23-5 shows the four available configurations:

(1) One process feeding one process

(2) One process feeding multiple processes

(3) Multiple processes feeding one process

(4) Multiple processes feeding multiple processes

For inter-process communications across the LAN, the *GBP* provides a set of relay and receiver programs that make communications virtually transparent to user programs, providing a simple and elegant approach to interfacing the Traffic Model Functions. See the next section for more information on queues sharing memory across the LAN.

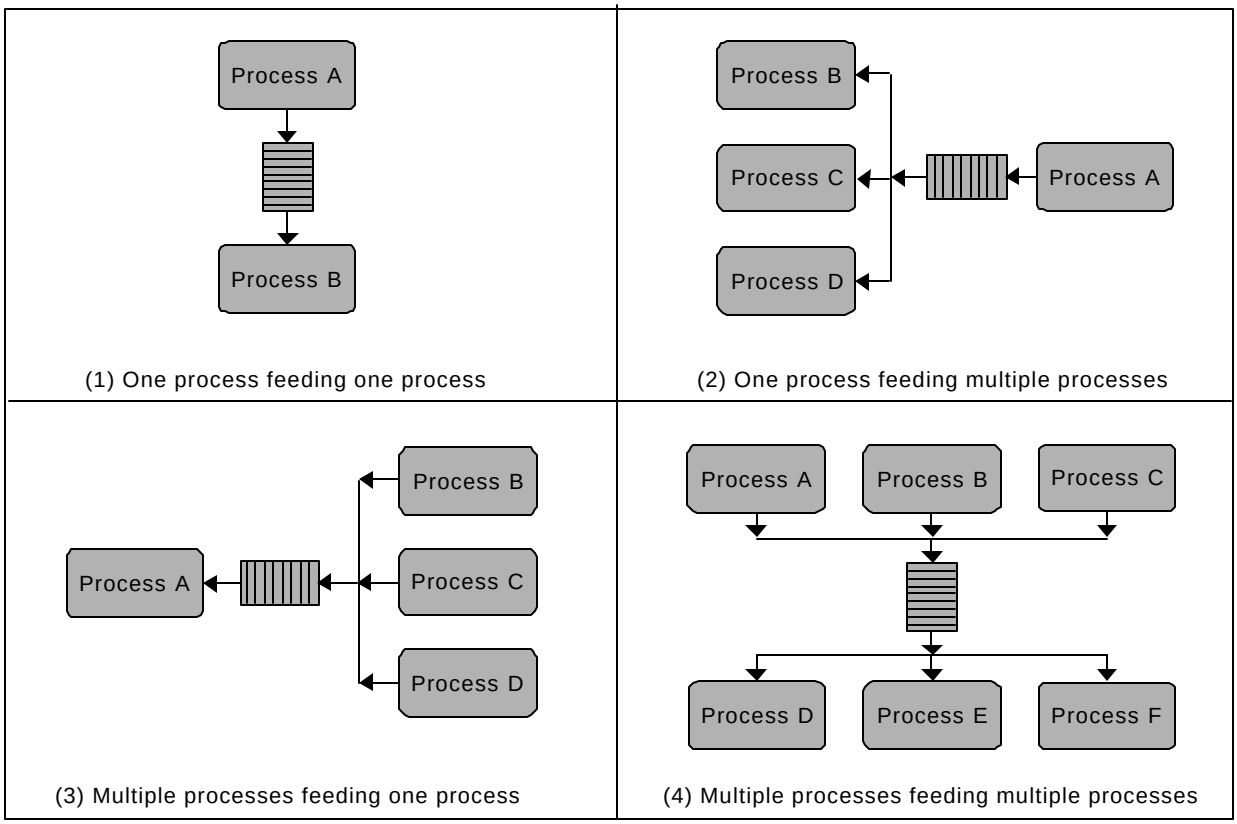

Figure 23-5. Buffering Data Flow Configurations

Implementing Shared Memory. In order to implement shared memory between otherwise independent processes, the virtual memory (VM) capabilities of the HP-UX system have been used extensively. See Section 4, under the heading *Mapped Files*.

A queue is a list of memory blocks, each of which contains a portion of the list of items in the queue. Memory blocks are mapped onto VM, and can be reached by any process with pointers to them. Access to any queue is provided by VM-mapped data structures that contain control information for each queue.

The memory blocks used in building any queue are dynamically managed by a global stack of free memory; that is, all queues using the same directory on a node share the same pool of memory blocks. Using a stack has the advantage of optimizing memory access, since the system uses demand-driven paged memory management. Once a memory block (one or several system memory pages) is popped from the stack, it will remain in use as long as possible, pushed in and popped from the stack, before the system resorts to another block. This translates into longer residency times for system memory pages in fast RAM, minimizing disk access.

Wherever there is data sharing, there has to be concurrency control in order to ensure data integrity; that is, a piece of data can't be modified simultaneously by two or more processes. UNIX semaphores are used to accomplish this.

Another issue that comes up when designing queue systems is the synchronization between enqueueing and dequeueing processes (for example, how to tell a dequeueing process that there are data in the queue, if it had checked before and the queue was empty). This aspect of inter-process communications has been addressed using eventcounts based on UNIX semaphores. General information on eventcounts can be found in Section 4.

In the queues case, eventcounts are used in two different instances:

(1) When a process is dequeueing data, and the queue is empty, it reads the current queue eventcount value, and then waits until the value is advanced by an enqueueing process.

(2) When a process is enqueueing data, and the queue has reached its maximum size, it reads the current queue block eventcount value, and then waits until the value is advanced by a dequeueing process, indicating a reduction of the queue size.

Eventcounts are also used in the stack routines for managing stack underflow. Stack underflow occurs when the free memory pool is temporarily depleted (i.e., when all memory blocks are currently in use, and the stack is empty). The popping routine then reads the stack eventcount and waits until it is increased by a push operation, indicating the release of a free memory block.

The interaction of locking and eventcount management, if not properly designed, may lead to deadlock situations. For example, a deadlock occurs when processes are waiting for eventcounts to advance, but the processes that advance the eventcounts are locked out of the data structures containing them, and are waiting for the locks to be released. Hence, all processes are in a waiting state at the same time. To prevent this undesirable situation, a dual *locking* mechanism has been developed, providing selective access to the stack and to each of the queues.

The *dual locking* mechanism provides a set of three locks for the stack as well as each queue. Each set includes the following locks:

- A global lock for the overall data structure. If this lock is set, no other process can access the structure.
- A lock for insert operations on the data structure. This lock applies to the stack operation *push* and the queue operation *enqueue*. If this lock is set, any process trying to insert data into the structure is locked out.

- A lock for retrieval operations on the data structure. This lock applies to the stack operation *pop* and the queue operation *dequeue*. If this lock is set, processes trying to fetch data from the structure are locked out.

Dual locking works as follows. Each stack or queue operation starts by trying to get the lock for that operation, before trying to get the global lock. Then, prior to entering a waiting state, the lock for the global structure is released, while the lock for the type of operation currently intended remains set. Hence, the only process that can get a lock for the structure is one performing the opposite operation, which then would cause the original operation to continue after the proper eventcount increment.

Queues in a Local Area Network Environment. A set of relay and receiver programs in the GBP expand the capabilities of this buffering package beyond a single node, and onto the LAN. Moreover, this is achieved in a manner almost totally transparent to the user programs, which don't have to deal with the communications details of it or be tied up waiting for I/O to complete.

When interdependent processes are running on different nodes, the *logical* data transfer queue can span several nodes, with relays and receivers handling the LAN inter-process communications. Figure 23-6 depicts the special cases, this time showing processes running on separate nodes. At the dequeueing end of some queues there are relays, sending data from local queues to receivers in other nodes, each connected at the enqueueing end of a local queue.

LAN inter-process communications is done via UNIX sockets. A *receiver* is a server, and hence can accept, and enqueue locally, input data from multiple channels. A *relay* is a client that opens a channel to a receiver, and sends locally dequeued data through the opened channel.

Having the *receiver* be the server allows for the possibility of multiple remote inputs into a local queue. By contrast, a *relay* has only one source of input, and needs to send data to only one destination (that is, the extension of the local queue), making it the ideal client. Moreover, data from a local queue may be processed in different places and can be distributed via multiple relays operating simultaneously on the same queue.

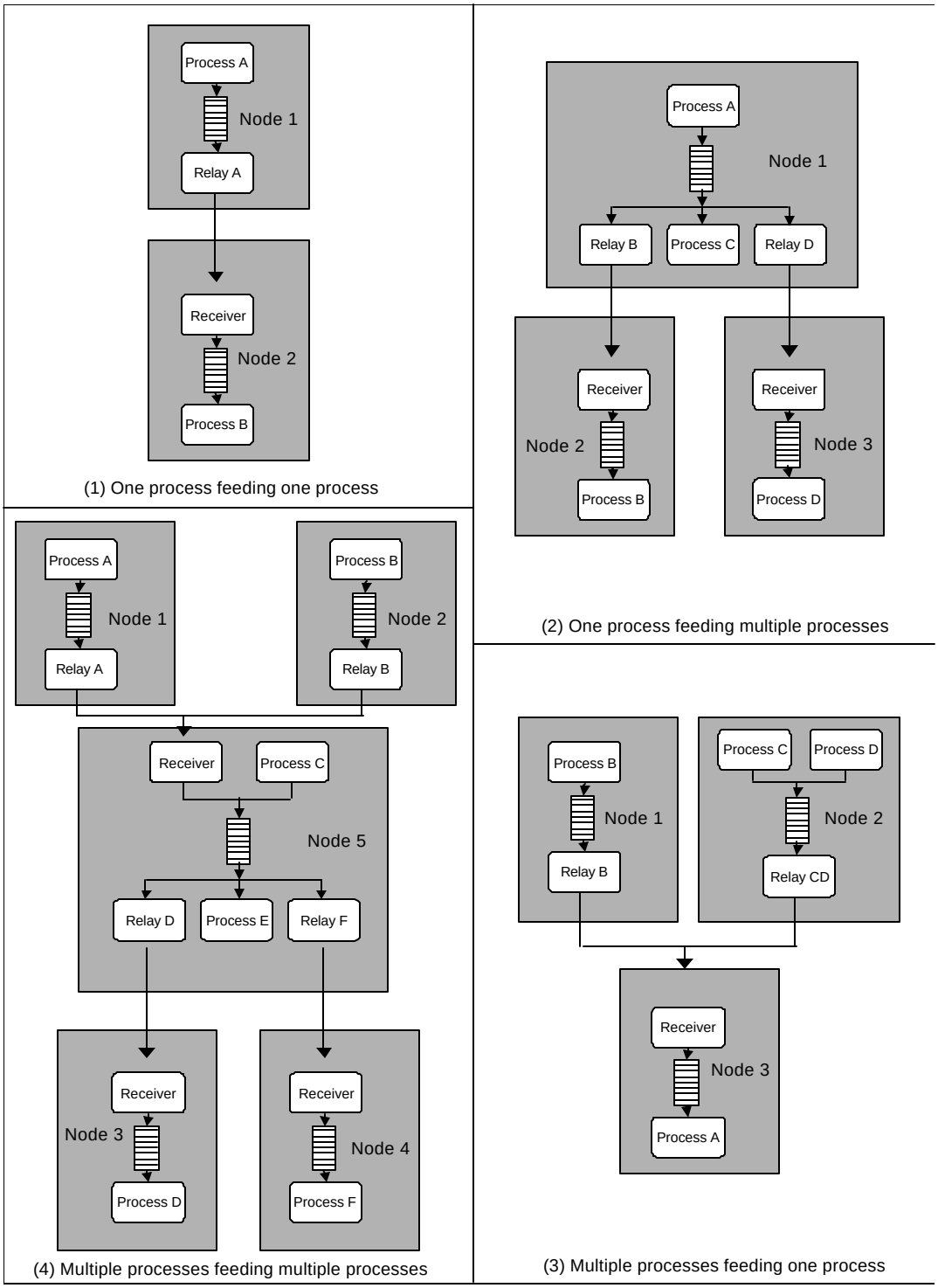

Figure 23-6. Internode Buffering Data Flow Configurations

For user programs, interfacing reduces to creating or opening a queue served by a relay in one end and a receiver at the other end. The following cases show generic LAN inter-process communications initialization logic for user programs whose queue data flows extend beyond the local node:

- If dequeueing, either try to hook up to a receiver that is already running, or start up a receiver. In either case queue identification information is collected from a *synchronization file*, which contains shared queue control information for the receiver and dequeueing processes.
- If enqueueing, either open or create a queue, and start up a relay process, passing the queue identifier and the TCP/IP socket filename as program parameters.

After the proper initialization steps, user programs enqueue and/or dequeue data items from the appropriate queues, while LAN inter-process communications are handled by relays and receivers.

Design Issue: Customizing the GBP

As with any generic package, the *GBP* needs to be customized for each usage. The customization is performed by parameters controlling the creation and management of the queues. The parameters are defined through an include file when the *GBP* is compiled.

The parameters file contains the following customized constants and type declaration:

- Maximum number of queues allowed
- Maximum number of queue items per queue block
- Total number of blocks in the free memory pool
- The data structure definition for items in the queue

The *GBP* also provides flexibility in the types of queue items it can handle using the same underlying data structures. Although in any customization there has to be only one data type definition for queue items, queues containing variable-size items can also be created simultaneously. This feature proves especially useful for processes handling multiple queues that contain different kinds of data.

When queues operate with variable-size items, these items are treated as character strings copied to and from the queue, provided a single item doesn't exceed the maximum data size for a block (size of declared fixed-size item times maximum number of items per block); i.e., no item can span multiple queue blocks.

Even though they share the same overall data structures, the fixed-record and variable-record queues are incompatible. Therefore, each case has separate subroutine calls, which cannot be

used interchangeably.

Processing Overview

The GBP is contained in three program modules to be bound with user programs, and two sets of programs containing the LAN inter-process communications components. The modules and sets of programs are described in the following sections.

23.2.2 Program Modules

23.2.1.1 The Server Module

Purpose

This module contains the enqueueing subroutines, *enqueue* and *varenqueue*, that perform fixed-size and variable-size record enqueueing operations, respectively.

Input

Both *enqueue* and *varenqueue* require as input the queue number, a unique identifier for the queue to enqueue data in.

In addition, *enqueue* needs as input the item to enqueue, which must be of the data type declared in the GBP instantiation include file. *Varenqueue* requires as input a pointer to the item to enqueue, and the length of the item.

Output

The only output of either *enqueue* or *varenqueue* is a boolean value, returning **true** if the enqueueing operation was performed successfully, **false** otherwise.

Processing

Processing follows similar threads for fixed- and variable-size queue items, differing only in the data copied into the queue. After getting the enqueue lock and the global queue lock, the module checks for available space in the last queue block and, if the check was unsuccessful, a new queue block is popped from the stack of free memory. If current queue size has reached its limit, then the global queue lock is released, and the process enters a wait state until the queue size decreases, when the global queue lock is grabbed again. Then, the enqueue lock is released, the item is copied into the current slot in the current queue block, and control returns to the caller after the global queue lock has been released.

Error Conditions and Handling

Following is a list of error conditions that cause either *enqueue* or varenqueue to return a status of **false**.

(1) Queue number provided has an invalid value.

(2) Queue with the number provided is inactive.

(3) Timeout waiting for a queue lock.

(4) Error occurred while waiting for a queue eventcount to advance.

23.2.1.2 The Client Module

Purpose

This module contains the dequeueing subroutines, *dequeue* and *vardequeue*, that perform fixed-size and variable-size record dequeueing operations, respectively.

Input

Both *dequeue* and *vardequeue* require as input the queue number, a unique identifier for the queue to enqueue data in.

In addition, *dequeue* requires as input a variable of the record type defined in the *GBP* instantiation include file to serve as placeholder for the item to be dequeued. *Varenqueue* requires as input a pointer to a buffer to serve as placeholder for the dequeued item, as well as a variable containing on entry the size of the buffer provided.

Output

Both *dequeue* and *vardequeue* return a boolean value as completion status, **true** if successful, **false** otherwise. If successful, both *dequeue* and *vardequeue* return the dequeued item in the placeholder provided on input. In addition, *vardequeue* returns the actual size of the item in the variable containing the buffer size on input.

Processing

Processing in the dequeueing subroutines follows the reverse thread of the enqueueing subroutines. After getting the dequeue lock and the global queue lock, the client module checks for items in the queue. If the check is unsuccessful, the global queue lock is released, the client module waits until an item is inserted in the queue, and the global queue lock is grabbed again. The item is then copied into a buffer area provided by the user, after the dequeue lock has been released. Once the item has been copied, the state of the current queue block is checked and, if the end of the block has been reached, the client module releases the block by pushing it into the stack of free memory. Finally, the global queue lock is released

and control returns to the caller.

Error Conditions and Handling

The following is a list of error conditions that cause either *dequeue* or *vardequeue* to return a status of **false**.

- Queue number provided has an invalid value.
- Queue with the number provided is inactive.
- Timeout waiting for a queue lock.
- Error occurred while waiting for a queue eventcount to advance.

23.2.1.3 The Support Module

This module contains all initialization and memory management routines, which include the stack operations. The following are routines contained in this module:

- Routines for queue and stack creation, opening and closing.
- Routines for stack and queue cleanup after abrupt program termination.
- *Hook_receiver*, a routine that hooks up to a receiver queue, starting a new receiver if necessary.
- The stack operation *push,* which releases a queue block already used, returning it to the free memory pool.
- The stack operation *pop*, which grabs an available memory block from the free memory pool.
- *Dequeue_ec_trigger*, a routine that returns the queue eventcount trigger value indicating the queue is not empty.
- *Check_queue*, a routine that prints queue status data to standard output for the queue with the number provided.

23.2.2 LAN Inter-process Communications Programs

The following sections briefly describe the *Receiver* and *Relay* communications programs.

23.2.2.1 The Receiver Processes

Purpose

Receivers receive data via the **Receiver** socket one item at a time, creating the need for two software components, to handle fixed- and variable-sized record enqueueing separately. The software components are called, respectively, *Decoupled Receiver* and *Fixed Receiver*. Note that the only difference between these two software components is the use of either *varenqueue* or *enqueue* subroutine calls.

Execution Control

A *Receiver* process is usually started up by a dequeueing process, but it can also be started at node startup or by the *nodescan* process.

Input

The *Receiver* processes require two parameters, with an optional third, to be provided in the program command line:

(1) The *Receiver* TCP/IP socket file name.

(2) The queue synchronization file name. This file is used to pass the queue number to all dequeueing processes which may access it.

(3) An optional parameter, containing the maximum number of queue blocks to allocate to the queue. By default, this parameter is set to 1000.

During processing, the Receiver processes accept three types of input, all in the form of messages:

(4) Channel open requests, which currently can be of two types:

  (a) Data input channels, which provide the *Receivers* with data to enqueue in the local queue.

  (b) Input request channels, which request from the *Receivers* the relay of data sent by the data input channels.

(5) Channel close notifications, from clients with open *Receiver* channels.

(6) Data messages, intended to be enqueued in the local queue by the *Receiver* processes and, if applicable, relayed to all the current input request clients to the *Receiver* mailbox.

Output

Output from the Receiver processes can be of two kinds:

(1) Control messages. These messages are sent by the *Receiver* processes to their clients in the following instances:

(a) When a client requests to open a channel connection, the *Receiver* process sends a message accepting or rejecting the request.

(b) A *Receiver* process may deallocate a client channel on request or as part of a cleanup routine before it aborts processing.

(2) Data messages. These messages, received from data input channels, are echoed to all the current input request channels, as well as enqueued in the local queue serviced by the *Receiver* process.

Processing

At initialization time, the *Receiver* process reads the program parameters in the command line and creates the *Receiver* socket, the queue synchronization file, and the local input data queue. If the *Receiver* process fails to create the synchronization file, it tries to open the synchronization file and open the local input data queue, if there is one already there; otherwise, it creates the local input data queue. After it finishes the initialization steps, the *Receiver* process advances an eventcount in the synchronization file, which signals to all dequeueing processes the availability of the local input data queue.

The *Receiver* process is a server that accepts two types of channels: data and input request channels. It waits for data to come via the data channels, echoes it to all the input request channels, and enqueues it in a local queue.

NOTE: A variation on the *receiver* idea allows parallel processing strings from a single data thread, each having the same input. A *tapping receiver* is a receiver that, instead of being a mailbox of its own, becomes an *input request* client to another *receiver*, which allows it to get all the data the *receiver* gets.

Error Conditions and Handling

The following is a list of error conditions causing program termination:

(1) No TCP/IP file name, or synchronization file name, in the program command line.

(2) The *Receiver* process failed to create or open the synchronization file with the name provided.

(3) The *Receiver* process failed to decode the optional command line parameter indicating the maximum number of queue blocks to allocate for the queue.

(4) The *Receiver* process failed to complete program initialization successfully. This error condition aggregates the following errors:

(a) The *Receiver* process failed to create the *Receiver* socket.

(b) The *Receiver* process failed to create or open the local data queue.

(c) The *Receiver* process failed to advance the synchronization file eventcount.

Error conditions that don't provoke process termination are listed below:

(5) If the *Receiver* process fails to send a message to a *Receiver* client, it checks the status code returned by the operating system. If the failure is due to the channel buffer being full, the process ignores it. Otherwise, the process deallocates the channel in question.

(6) The *Receiver* process ignores any messages that are neither data, open requests, nor channel close notifications.

(7) If the *Receiver* process fails to enqueue data received from the *Receiver* socket, it prints an error message to standard output and continues processing. No further buffering of the data that was to be enqueued is done, thus effectively losing the data.

23.2.2.2 The Relay Processes

Purpose

The *Relay* processes handle queue data transmissions across the LAN. Depending on the format of the queue data transmitted across the LAN, there are two types of Relay processes:

(1) *Fixed Relay*. These *Relay* processes handle the transmission of fixed-record size queue data.

(2) *Variable Relays*. These *Relay* processes handle the transmission of variable-record size queue data.

Execution Control

A *Relay* process is usually started up by an enqueueing process, but it can also be started at node startup or by the *nodescan* process.

Input

The Relay processes require as input the following command line parameters:

(1) Parameters file name. The parameters file contains the following program parameters:

(a) *Receiver* TCP/IP file name, followed by a *retry* indicator.

(b) *Coupled/Decoupled* communications indicator.

(c) *Receiver* socket client type name.

(d) Free format text to append to the *Receiver* socket open request, to be displayed by the *Receiver* process upon acceptance of the open request.

(2) Queue number, a unique identifier for the output data queue.

During processing, the *Relay* processes get their only input from the output data queue.

Output

The output of the *Relay* processes is data from the output data queue, transmitted to the *Receiver*.

Processing

At initialization time, a *Relay* process reads the parameters file name and the output data queue number from the program command line. The *Relay* process then reads the *Receiver* socket name from the parameters file. If there is a socket file name in the parameters file, the *Relay* opens a channel to the *Receiver*. Otherwise, the *Relay* sets a flag indicating no *Receiver* connection is requested. Two more flags are set by the *Relay* at initialization: a flag for retrying to open *Receiver* after losing the connection, and a flag indicating either coupled or decoupled communications mode.

A relay is a dequeueing process that becomes a data input client to a *Receiver*. The normal processing sequence for a *Relay* process consists of an endless iteration between dequeueing data from the data output queue and transmitting the data to the *Receiver* socket, if the proper flag is set.

Error Conditions and Handling

The following is a list of error conditions causing termination of a *Relay* process:

(1) No parameters file name provided in the command line, or the *Relay* process was unable to open the file provided.

(2) No output data queue number provided in the command line, or the *Relay* process was unable to open the queue with the number provided.

The following is a list of recoverable error conditions for the *Relay* processes:

(3) If the *Relay* process is not able to connect to the *Receiver*, further processing depends on the status of the *retry* flag. If it is set, the *Relay* process will keep trying to connect to the *Receiver* until it succeeds. If the retry flag is not set, the *Relay* process will enter normal processing, after disabling all further data transmissions.

(4) The *Relay* process receives an operating system error status after trying to transmit data to the *Receiver*, and the status code doesn't match any of the special cases mentioned above. In this case, the *Relay* process closes the *Receiver* channel and, if the proper flag is set, it tries to reconnect to the *Receiver*. Upon successful reconnection to the *Receiver*, the *Relay* process retransmits the data item to the *Receiver*.

(5) If a call to the dequeueing routine (*dequeue* or *vardequeue)* fails, the *Relay* process will print the output data queue and data transmission statistics to the standard output stream, skipping the data transmission step, since there is nothing to transmit.

23.2.3 Generic Buffering Package Data Structures

The *GBP* data structures are divided into three main groups:

(1) Those visible to the user, needed for access to the queueing facilities

(2) Internal data structures supporting the *GBP*

(3) Those used for customization of the package. Since definitions for these structures differ by customization, they are not described here.

23.2.3.1 User-accessible Data Structures

The following are global filenames and variables that provide the user with limited access to different aspects of the package.

23.2.3.1.1 Filenames

- **Stackfile** - mapped file containing the memory pool control stack, and queue control data. The default is **memory_stack_and_queues**, in the current working directory.
- **Blockfile** - mapped file of free memory blocks. The default is **memory_blocks**, in the current working directory.
- **Receiver_synchro** - synchronization file for starting inter-process queue operations. The user *must* set this name if it wants to use this file, since there is no default.
- **Receiver_mailbox** - TCP/IP file name for incoming messages to local receiver. The user *must* set this name if it is a receiver.

23.2.3.1.2 Other Globals

- **Synchro_ptr** - auxiliary variable that may be used to point at synchronization file mapped in virtual memory.
- **Queue_stack** - pointer to start of *Stackfile* in virtual memory.
- **Blockfilestart** - starting offset of mapped *Blockfile* in virtual memory.
- **Enqueue_eventcounts** - array of pointers to eventcounts for each queue. Can be used for selective waits in programs with more than one input, triggering a call to *dequeue* or *vardequeue.*
- **Wait_time_for_lock** - maximum time to wait for a queue or stack lock to be released, before returning error status.

23.2.3.1.2 Synchronization Data Type

Table 23-5 illustrates the structure of the synchronization table, which contains shared queue control information for the receiver and dequeueing processes.

Table 23-5. Synchronization_record_t Data Structure

Synchornization_record_t			
Library Name: buffer_openlib		**Element Name**: buffer.h	
Purpose: Serves as synchronization structure for interprocess communications (IPC) using queues. It stores the quiue id, and controls startup of communications.			
Data Item	**Definition**	**Range**	**Var. Type**
eventcount	counter controlling quiue status	EVT_T	
aueuenumber	IPC quiue id	0..maxqueues	queue_range
severity	severity returned from process creating queue	severity levels (0..15)	short

23.2.3.2 Internal Data Structures

The remaining tables in this section illustrate data structures that are used within the *GBP* itself to manipulate stacks and queues.

Table 23-6. Buffer_control_record_t_Data Stucture

Buffer_control_record		
Library Name: buffer_openlib	**Element Name**: internals.h	
Purpose: GBP structure for dynamic memory measurement		
Data Item	**Definition**	**Variable Type**
queuelist_semaphore		int
mkey		int
next_key		int
last_free_block		int
front_of_list		int
blocks_allocated		int
max_blocks_allocated		int
q_list	array of queues	queue_control_record

Table 23-7. Queue_control_record_t Data Sructure

queue_control_record		
Library Name: buffer_openlib	**Element Name**: internals.h	
Purpose: Contains control structures for a single queue		
Data Item	**Definition**	**Variable Type**
eventcount		EVT-T
queue_semaphore		int
front_of_q		int
back_block_num		int
back_of_q		int
entries_enqueued		int
entries_dequeued		int
blocks_limit		int
blaock_in_use		int
total_blocks_used		int
max_blocks_used		int
active		Boolean
use_evt		Boolean